UMAR TARIQ
01-134221-085

Urban Waterz

Bachelor of Science in Computer Science

Supervisor: Dr. Arif Ur Rahman

Department of Computer Science
Bahria University, Islamabad

December 2025

Certificate

We accept the work contained in the report titled "Urban Waterz", written by Mr. Umar Tariq as a confirmation to the required standard for the partial fulfillment of the degree of Bachelor of Science in Computer Science.

Approved by

Supervisor: Dr. Arif Ur Rahman

_____

Internal Examiner: Name of the Internal Examiner (Title)

_____

External Examiner: Name of the External Examiner (Title)

_____

Project Coordinator: Name of the Project Coordinator (Title)

_____

Head of the Department: Name of the HOD (Title)

_____

December 2^{nd}, 2025

Abstract

Reliable access to water is a fundamental requirement for households and businesses, particularly in urban areas where municipal water supply is irregular or insufficient. In many such localities, residents depend on water tankers and overhead storage tanks. Traditional processes for ordering water are manual: customers typically place orders via phone calls or messaging apps, track deliveries informally, and monitor tank levels by guesswork. This results in uncertainty, service delays, and inefficient resource utilization. This final year project presents Urban Waterz, an IoT-enabled water tank monitoring and tanker dispatch system. Urban Waterz integrates four main subsystems: An ESP32 IoT node with ultrasonic and DHT sensors monitors tank level and environmental conditions. A Node.js/Express backend with MongoDB and Socket.IO handles storage and real-time communication. An Expo React Native app provides role-based interfaces for customers and drivers, while a React web app offers admin dashboards for user, order, and driver management, along with system analytics. Sensor readings from the IoT node are sent securely to AWS IoT Core using MQTT. The backend subscribes to these messages, applies a calibration-based formula to compute the water tank level as a percentage, and stores the result in a dedicated IoT data collection. Customers can view real-time tank level and historical usage trends, and can place orders for large or small tankers and water bottles through the mobile application. Admins use a web dashboard to manage customers, drivers, and orders, with features such as driver assignment, driver queues, user blocking, and statistics. Drivers receive assigned orders, update statuses, and share their location via the driver app. Socket.IO ensures that order status, driver status, and driver location updates propagate in real time to all stakeholders. The project demonstrates how a full stack architecture can unify physical sensing and logistical operations into one coherent platform. The system has been designed with a modular architecture using well-structured models and services, making it extensible for future features such as payment integration, predictive analytics, and multi-tenant support. The evaluation indicates that Urban Waterz improves visibility, coordination, and user experience compared to traditional manual systems, and provides a solid foundation for real deployments in urban water management.

Acknowledgements

We would like to express our deepest gratitude to Allah Almighty, whose countless blessings and guidance have enabled us to successfully complete this project. Without His mercy, this accomplishment would not have been possible. We extend our sincere appreciation to our supervisor, Dr. Arif Ur Rahman, for his constant support, valuable guidance, and encouragement throughout the course of this project. His insights and feedback greatly shaped the direction of our work and helped us overcome many challenges. We are also thankful to our department, the Bahria University, Islamabad, for providing us with the resources and learning environment needed to explore, innovate, and implement our ideas effectively. Our heartfelt thanks go to our families and friends, whose continuous encouragement, patience, and prayers gave us the strength to remain focused and motivated during this journey. Finally, we acknowledge the collaborative efforts of our entire team. Each member's dedication, hard work, and contribution were instrumental in completing the IoT-Based Smart Parking Management System successfully.

UMAR TARIQ
Islamabad, Pakistan

December, 2025

"When a man says I cannot, he has made a suggestion to himself. He has weakened his power of accomplishing that which otherwise would have been accomplished"

Muhammad Ali

Contents

List of Figures

List of Tables

Acronyms and Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CORS	Cross-Origin Resource Sharing
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DHT	Digital Humidity and Temperature
ER	Entity-Relationship
EST	Eastern Standard Time
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IoT	Internet of Things
JSON	JavaScript Object Notation
JWT	JSON Web Token
KPI	Key Performance Indicator
MQTT	Message Queuing Telemetry Transport
NoSQL	Not Only SQL
NTP	Network Time Protocol
PDF	Portable Document Format
RBAC	Role-Based Access Control
REST	Representational State Transfer
SDK	Software Development Kit
SMS	Short Message Service
UI	User Interface
UML	Unified Modeling Language
UPS	Uninterruptible Power Supply
URL	Uniform Resource Locator
USB	Universal Serial Bus
XML	Extensible Markup Language

Chapter 1

Introduction

Water scarcity, inconsistent municipal supply, and increasing reliance on water tankers have created significant challenges for households, businesses, and tanker service providers in metropolitan areas. Most users depend on overhead or underground tanks for daily storage, yet tank levels are often estimated manually, leading to unexpected shortages, inefficiencies, and mismanagement. At the same time, tanker suppliers struggle with uncoordinated orders, unclear demand patterns, and limited visibility into driver availability and performance.

Urban Waterz is an integrated IoT-enabled water monitoring and dispatch management system designed to address these challenges. By combining real-time IoT tank monitoring with an end-to-end tanker ordering and delivery workflow, Urban Waterz enhances operational transparency, reduces inefficiencies, and improves the user experience across customers, drivers, and administrators. Leveraging ESP32-based IoT nodes [1], cloud communication, a unified backend, and dedicated mobile and web applications, the system creates a seamless, data-driven approach to water management.

1.1 Problem Statement

In many urban locations, homes and businesses often face sudden water shortages because tank levels are not accurately monitored. People frequently guess tank levels by looking inside the tank or based on routine usage. This leads to situations where water finishes unexpectedly, especially during peak hours. When this happens, users urgently attempt to order a tanker—usually by calling

suppliers or sending WhatsApp messages—resulting in delays, confusion, or missed deliveries.

On the supplier side, dispatchers lack proper visibility into incoming orders, driver availability, or delivery progress. Drivers receive instructions informally and without proper tracking, leading to inefficiencies and customer dissatisfaction. The absence of real-time communication and historical data further complicates planning and optimization.

To address these challenges, the proposed solution is a unified IoT-Based Water Tank Monitoring and Tanker Dispatch System. This system integrates real-time tank level monitoring with automated order handling, driver coordination, and administrative visibility, providing a complete digital solution from measurement to delivery.

1.2 Stake Holders

- Homeowners and businesses relying on water tanks
- Water tanker drivers
- Water tanker service providers
- IoT hardware providers
- Software development and support teams
- Administrative authorities using the system
- Organizations handling billing and taxation
- Cloud and infrastructure service providers

1.3 Project Objectives

The primary objective of the Urban Waterz system is to design and implement an IoT-enabled platform that ensures uninterrupted water availability through real-time monitoring and structured delivery management. The system focuses on eliminating common issues such as sudden water depletion, order mismanagement, driver delays, and manual payment inefficiencies.

More specifically, Urban Waterz aims to:

- Provide accurate, real-time monitoring of water tank levels using an ESP32 IoT module.
- Enable customers to place tanker and bottle orders through a mobile application.
- Streamline order handling through a managed order lifecycle.
- Allow administrators to assign drivers, monitor deliveries, and track system status.
- Provide real-time updates and communication using Socket.IO [2].
- Improve reliability, transparency, and efficiency for both customers and service providers.

1.4 Scope

This project focuses on developing a fully functional IoT-based water management system with the following core components:

- A physical IoT node (ESP32 with ultrasonic and DHT sensors) installed on a test water tank.

- A backend integrating AWS IoT Core [3] and MongoDB to handle device data, orders, users, and drivers.
- A mobile application for customers and drivers with separate interfaces for ordering, tracking, and navigation.
- A web-based admin dashboard for managing drivers, viewing IoT data, and monitoring system performance.
- Real-time data synchronization across all platforms using Socket.IO.

Future enhancements such as payment gateway integration, large-scale deployment, AI-based demand forecasting, and route optimization are outside the scope of the current implementation but may be explored in future work.

Chapter 2

Review of Literature

This chapter presents a comprehensive review of existing literature and technologies relevant to the Urban Waterz system. The review is organized into distinct areas that correspond to the core components and functionalities of the project: IoT-based monitoring systems, on-demand delivery and dispatch platforms, and integrated systems that combine both sensing and service delivery.

The primary objective of this literature review is to establish the context and rationale for Urban Waterz by examining prior work in water tank monitoring, delivery logistics, and hybrid IoT-service platforms. By analyzing the strengths and limitations of existing solutions, this chapter identifies the gaps that Urban Waterz addresses—specifically, the lack of unified systems that integrate real-time tank monitoring with tanker dispatch operations.

Section 2.1 explores IoT-based monitoring systems, focusing on sensor technologies, microcontroller platforms, and cloud integration methods commonly used for water level measurement. Section 2.2 examines on-demand delivery and dispatch systems, drawing parallels from ride-hailing and food delivery applications to understand user role management, order lifecycle patterns, and real-time tracking mechanisms. Section 2.3 discusses emerging integrated IoT-service platforms that combine physical sensing with operational decision-making. Finally, Section 2.4 provides a comparative analysis that positions Urban Waterz relative to existing solutions, highlighting its unique contributions to the domain of urban water management.

2.1 IoT-Based Monitoring Systems

IoT (Internet of Things) technologies are widely used for sensing and monitoring environmental variables such as temperature, humidity, distance, and fluid levels in storage systems. Commonly used components include DHT sensors for temperature and humidity, ultrasonic sensors for distance measurement, and microcontrollers such as Arduino Uno, ESP8266, and ESP32 [4].

Most existing water tank monitoring systems use ultrasonic sensors mounted at the top of a tank to measure water depth and send data to a cloud server or dashboard. These systems typically focus on sensing and visualization, with limited emphasis on user accounts, multi-role functionality, or operational integration.

Key characteristics of typical IoT monitoring solutions include:

- Focus on sensing and data display rather than operational workflows.
- Minimal features for user management or role-based access.
- No integration with downstream services such as ordering or dispatch.

Urban Waterz builds on these sensing principles but connects tank-level monitoring to real-world operations such as placing orders, dispatching drivers, and managing deliveries.

2.1.1 On-Demand Delivery and Dispatch Systems

On-demand delivery platforms follow standardized patterns that enable efficient service fulfillment through structured workflows, role separation, and real-time coordination between stakeholders.

2.1.1.1 Multiple User Roles

Most delivery platforms operate using a role-based access control model that assigns specific functions and permissions to each type of user such as customers, drivers, and administrators.

- Consumer (Customer)
- Driver or service provider
- Admin / dispatcher

2.1.1.2 Order Lifecycle

Orders in delivery platforms move through a well-defined set of states that reflect progress from request to fulfillment.

- Request
- Assign / Accept
- In progress
- Completed / Cancelled

2.1.1.3 Driver State Management

Drivers operate in different states so that the system can correctly track their availability and assign deliveries efficiently.

- Available
- Busy
- Offline

2.1.1.4 Real-Time Communication

Live updates ensure that customers and administrators remain informed throughout the delivery process.

- Customers track driver movement.
- Drivers receive instant task updates.

2.2 Integrated IoT + Service Platforms

Several modern systems combine IoT data with service workflows to create responsive, intelligent platforms.

Examples include:

Smart logistics and cold chain systems

- Sensors track temperature inside delivery trucks.
- Alerts generated when thresholds are exceeded.
- Data used for quality control and routing decisions.

These examples demonstrate how IoT measurements influence operational decisions. Urban Waterz follows a similar principle by using tank-level data to:

- Notify customers when water levels drop (e.g., below 30
- Help administrators forecast demand.
- Identify high-usage customers based on data patterns.

2.3 Comparison with Existing Systems

This section provides a comparative analysis of the proposed system with existing solutions in the domain of water monitoring and delivery management. The comparison highlights functional limitations in existing systems and demonstrates how the proposed solution addresses these gaps.

2.3.1 Basic Tank Monitor

Basic tank monitoring systems are primarily designed to display the real-time water level of storage tanks using simple sensors. While these systems are

effective for monitoring purposes, they lack integration with water delivery services. They do not support order placement, delivery tracking, or role-based user access. Additionally, such systems do not provide administrative dashboards or data analytics capabilities. As a result, users must rely on external means to arrange water deliveries, making the overall process inefficient and fragmented.

2.3.2 Generic Delivery Application

Generic delivery applications focus on managing orders and deliveries, often supporting multiple user roles such as customers and drivers. These applications typically provide features such as order placement and real-time driver tracking. However, they are not specifically designed for water management scenarios and do not integrate tank-level data into their operations. Consequently, delivery decisions are not automated based on actual water availability, and administrative control varies depending on implementation. The lack of integration with IoT-based tank monitoring limits their effectiveness in water resource management.

2.3.3 Urban Water (This Project)

The proposed Urban Water system overcomes the limitations of existing solutions by integrating real-time tank level monitoring with an intelligent water delivery management platform. It supports multiple user roles, including customers, drivers, and administrators, each with clearly defined functionalities. The system enables automated order placement based on tank levels, real-time driver tracking, comprehensive administrative dashboards, and IoT data visualization. Furthermore, the modular and extensible design allows future enhancements without major system restructuring, making it a robust and scalable solution for urban water management.

Urban Waterz stands out by combining real-time IoT tank monitoring with a structured water delivery and dispatch system in a single integrated platform.

Table 1: Comparison with Related Work

Feature	**Basic Tank Monitor**	**Generic Delivery App**	**Urban Waterz (This Project)**
Real-time tank level	✓	×	✓
Orders / deliveries	×	✓	✓
Multiple user roles	×	✓	✓
Real-time driver tracking	×	✓	✓
Admin dashboards	Limited	Varies	✓
Tank level linked to operations	×	×	✓
Extensible modular design	Varies	Varies	✓

Chapter 3

Requirement Specifications

This chapter presents the requirement specifications that form the foundation of the Urban Waterz system. As Urban Waterz integrates IoT-based water tank monitoring with an on-demand water tanker dispatch and delivery platform, defining clear requirements is essential for ensuring consistent behavior across all system components. The requirements outlined here are based on real-world user needs, analysis of existing service gaps, and observations from traditional water-ordering and monitoring practices.

The chapter begins by explaining the goals and roles of all interacting Customers, Drivers, and Administrators. It then details the functional requirements that describe the core operations of the system, including real-time monitoring, order lifecycle management, driver assignment, and communication flows. Non-functional requirements related to performance, security, usability, scalability, and maintainability are also addressed to ensure the system performs reliably in real deployment environments.

3.1 Use-Case Modelling and Storyboarding

This section expands on the short descriptions given earlier and adds storyboards and event–response tables.

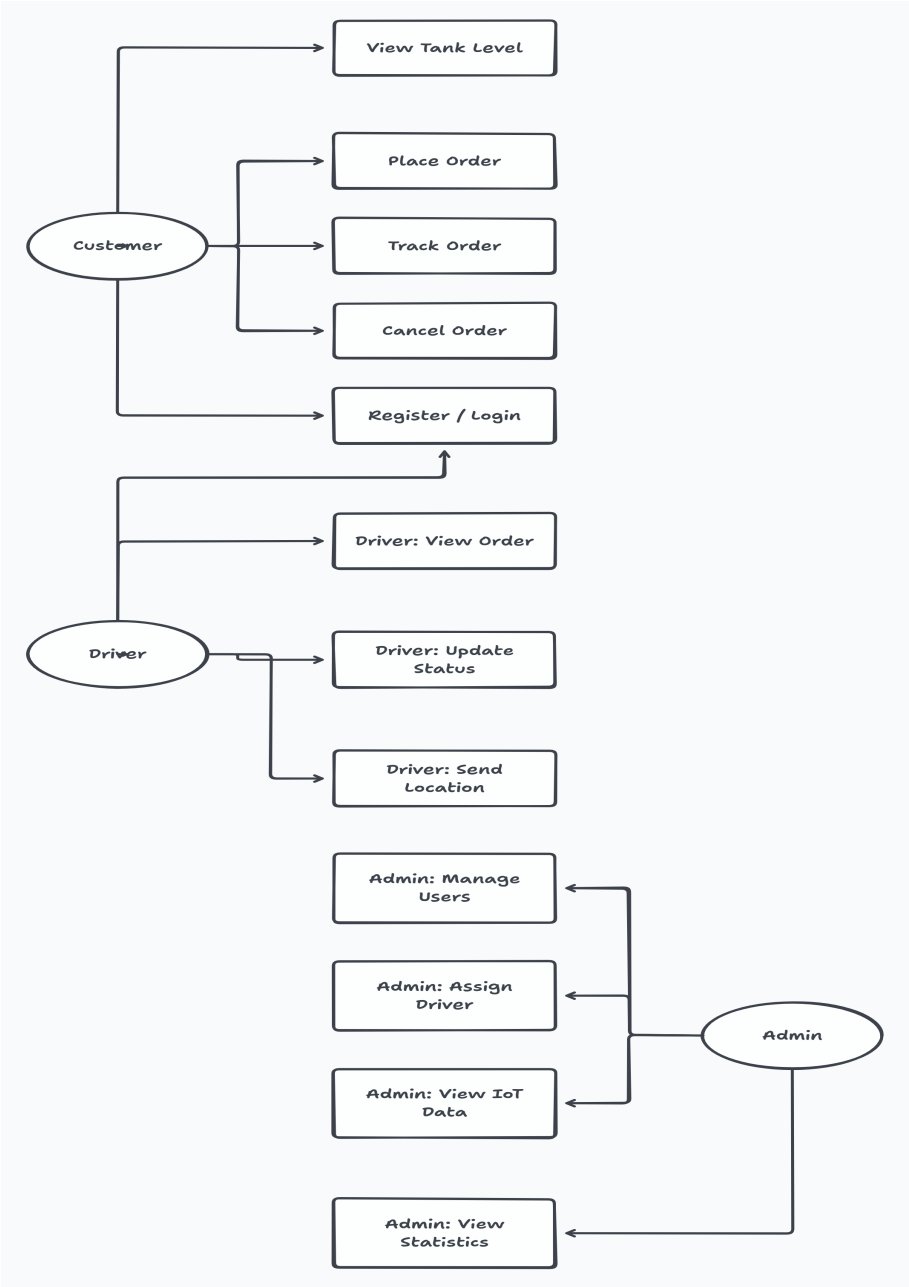

Figure 3.1: Use Case Model

3.1.1 Event–Response Table (Extended)

Table 2: Event–Response Table

Event	Source	System Response
Sensor reading with distance published	ESP32 firmware	IoT subscriber computes `tankLevel`, creates IoTData record, updates `latestData`.
Customer taps "Refresh Data" on tank tab	Mobile app	App calls `/iot/latest` and `/iot/all`, updates chart, alerts, and usage estimations.
New order created	Customer	Backend stores order, emits `new-order` to `admin-room`, returns order details to client.
Admin assigns order to driver	Admin dashboard	Backend updates `Order.driver` and driver queue; emits `order-assignment` and queue updates.
Driver completes order	Driver app	Backend updates order status to `delivered`, recalculates driver status and queue, emits events.
Admin blocks user	Admin dashboard	Backend updates `User.status` to `blocked`, records reason/timestamp, denies future logins.
Blocked user attempts login	Mobile/web app	Backend rejects with HTTP 401 and "User is blocked" message.
Driver updates GPS location	Driver app	Backend updates `User.location` and emits `driver-location-update` to admin and customer.

3.1.2 Screens Overview

This section outlines the primary user interfaces for each type of user in the system. Each interface is designed to ensure usability, clarity, and access to relevant functions based on role.

3.1.2.1 Customer Screens

Customer interfaces focus on monitoring water levels, placing orders, and tracking deliveries.

- Login / Sign-up Screen
- Home Dashboard
- Tank Monitoring Screen
- Order Creation Screen
- Orders List and Order Details

3.1.2.2 Driver Screens

Driver interfaces prioritize navigation, delivery updates, and earnings overview.

- Login Screen
- Dashboard
- Order Details Screen
- Earnings Screen

3.1.2.3 Admin (Web) Screens

Admin dashboards support system-wide monitoring and control.

- Login Page

- Main Dashboard
- User Management Page
- Order Management Page
- Driver Management Page
- IoT Monitoring Page

3.2 Functional Requirements

3.3 Functional Requirements

The system shall provide the following functional capabilities:

1. Allow new users to register with type: customer, driver, or admin.
2. Authenticate users using email/password and issue JWT.
3. Provide endpoints for users to view and update their profile information.
4. Allow users to change password.
5. List all available product types and their prices.
6. Customers can create orders specifying items, address, notes, and payment method.
7. Customers can view their orders and detailed information.
8. Drivers and admins can list all orders, filterable by status or driver.
9. Support allowed order status transitions only.
10. Customers can cancel orders when pending, confirmed, or preparing.
11. Admins can assign orders to drivers.

12. Maintain a driver queue for each driver.
13. Admins can reorder driver queues and remove orders.
14. Drivers can see current order and queue; mark orders delivered.
15. Drivers can set status to free, busy, or offline with validation.
16. Driver app sends GPS updates, saved and broadcast in real time.
17. IoT firmware sends sensor readings to backend via AWS IoT Core.
18. Compute `tankLevel` from sensor distance and calibration.
19. Store IoT readings in the database with timestamps.
20. Clients can retrieve latest and historical IoT readings (paginated).
21. Admins can set tank calibration parameters.
22. Admin APIs for listing, searching, and viewing users.
23. Admins can block/unblock users with reason.
24. Admins can change user types with validation.
25. Provide statistics of users by type/status and orders by status.
26. Generate notifications for key events; APIs to view and mark read.
27. Drivers can create support tickets and track their status.

3.4 Non-Functional Requirements

The system shall satisfy the following non-functional requirements:

- Security: Passwords hashed with strong algorithm (bcrypt).
- Security: JWT tokens signed with secret, expire after configurable time.

- Security: Socket.IO validates JWT, rejects blocked users.
- Performance: Support concurrent connections without significant delay.
- Performance: Endpoints support pagination for large datasets.
- Reliability: Backend uses robust error handling (global middleware, process-level handlers).
- Usability: Mobile/web UIs intuitive, responsive, with visual cues.
- Maintainability: Backend structured into models, services, controllers, routes.
- Portability: Mobile app runs on Android/iOS (React Native + Expo).
- Scalability: Architecture extensible to multiple IoT nodes/admin dashboards.

3.4.1 Assumptions and Constraints

This section identifies operating conditions that the system expects and highlights practical limitations that may affect deployment or performance.

3.4.1.1 Assumptions

The system operates on the premise that users and devices follow expected operating environments.

- Users have smartphones and active internet connections.
- IoT devices are installed correctly.
- AWS IoT Core is properly configured.
- MongoDB and Node.js are available.

3.4.1.2 Constraints

These limitations define boundaries within which the system must realistically operate.

- IoT hardware has limited power and memory.
- Network connectivity may be unstable.
- AWS free tier limitations.
- Academic time constraints restrict feature expansion.

Chapter 4

System Overview

Urban Waterz is designed with a multi-layered architecture integrating IoT devices for real-time water tank monitoring and order management.

4.1 Architectural Design

4.1.1 High-Level Architecture

The Urban Waterz system is designed as a layered architecture where each layer performs a specialized role in enabling monitoring, communication, and service execution.

4.1.1.1 IoT Device Layer

This layer is responsible for sensing water levels and environmental data.

- ESP32 microcontroller
- Ultrasonic sensor
- DHT sensor

4.1.1.2 Cloud Integration Layer

This layer ensures secure communication between devices and backend services.

- AWS IoT Core
- MQTT protocol
- Device certificates

4.1.1.3 Application Backend

Handles data processing, business logic, and API services.

- Node.js and Express
- Socket.IO
- MongoDB

4.1.1.4 Client Layer

Provides interfaces for different users.

- Mobile App (Customer / Driver)
- Web Dashboard (Admin)

4.1.2 Architecture Diagram

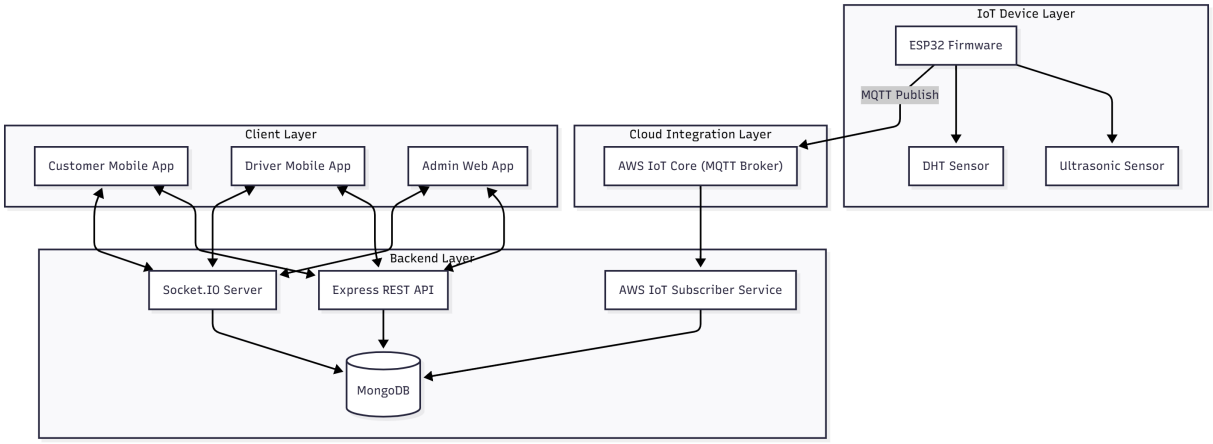

Figure 4.1: High-Level Architecture of Urban Waterz

4.2 Data Design

4.2.1 Logical Data Model

The logical data model defines how information entities are structured and related within the database.

4.2.1.1 User

The user entity represents any individual interacting with the system and stores authentication and profile-related data.

- userType (customer, driver, admin)
- Name, email, password, phone number
- Status and timestamps

4.2.1.2 Order

The order entity records all information related to water deliveries.

- Order ID and references
- Items and pricing
- Delivery address
- Order and payment status

4.2.2 ER Diagram

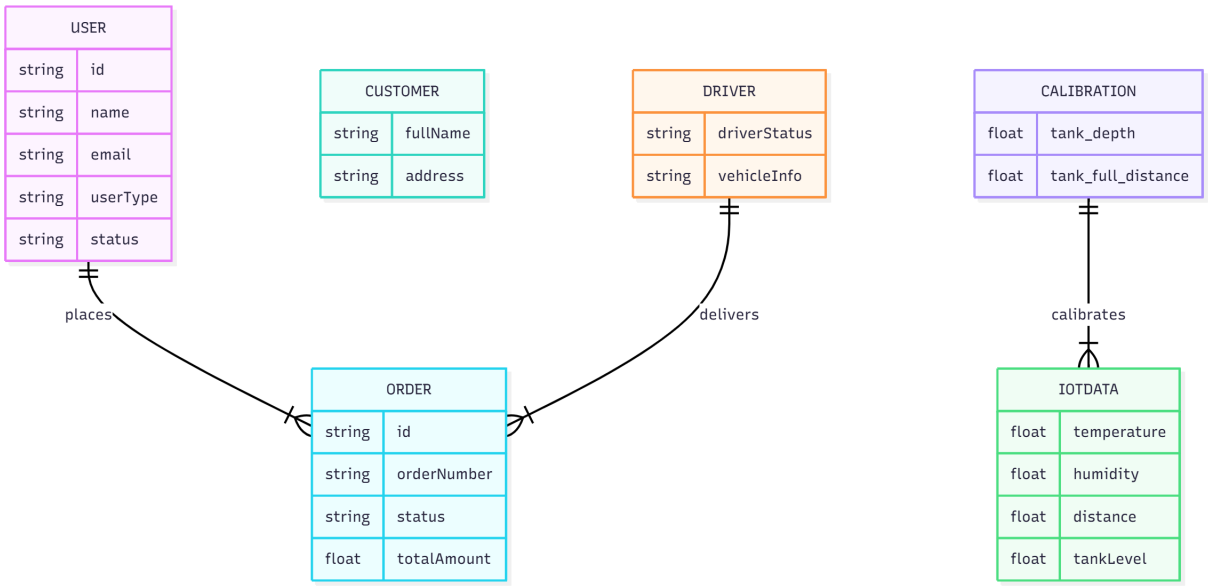

Figure 4.2: Entity-Relationship Diagram of Urban Waterz

4.3 Domain Model

The domain model captures conceptual entities and their relationships:

- **Person / User:** Human interacting with the system; attributes include name, email, and authentication credentials.
- **Customer:** User type with extended address info, places orders and monitors tank level.
- **Driver:** User type fulfilling orders, maintaining vehicle info and order queue.
- **Admin:** User with full access, manages other users and system operations.
- **Order:** Represents contract between customer and service, including items, delivery, and cost.
- **Tank:** Physical container storing water; state represented by Calibration + IoTData.
- **Sensor Reading:** Each IoTData entry is a snapshot of tank state at a particular time.

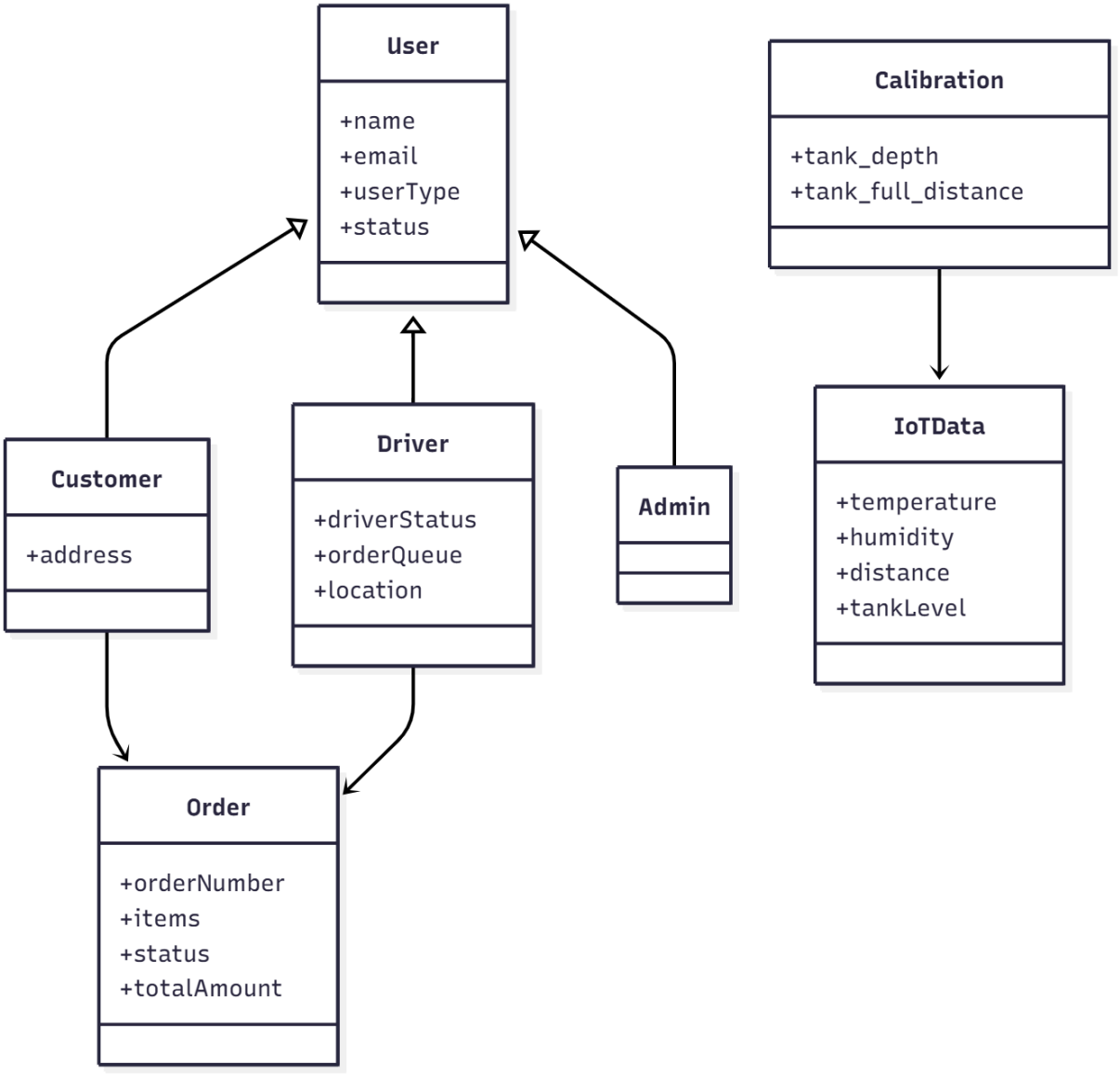

Figure 4.3: Domain Model of Urban Waterz

4.4 Design Models and Diagrams

4.4.1 Use-Case Model

This model identifies the actions each user role can perform within the system.

- **Customer**: Register, View Tank Level, Place Order, Track Order
- **Driver**: View Orders, Update Status, Update Location, View Earnings
- **Admin**: Manage Users, Assign Drivers, View Reports

4.4.2 Sequence Diagram – Order Assignment

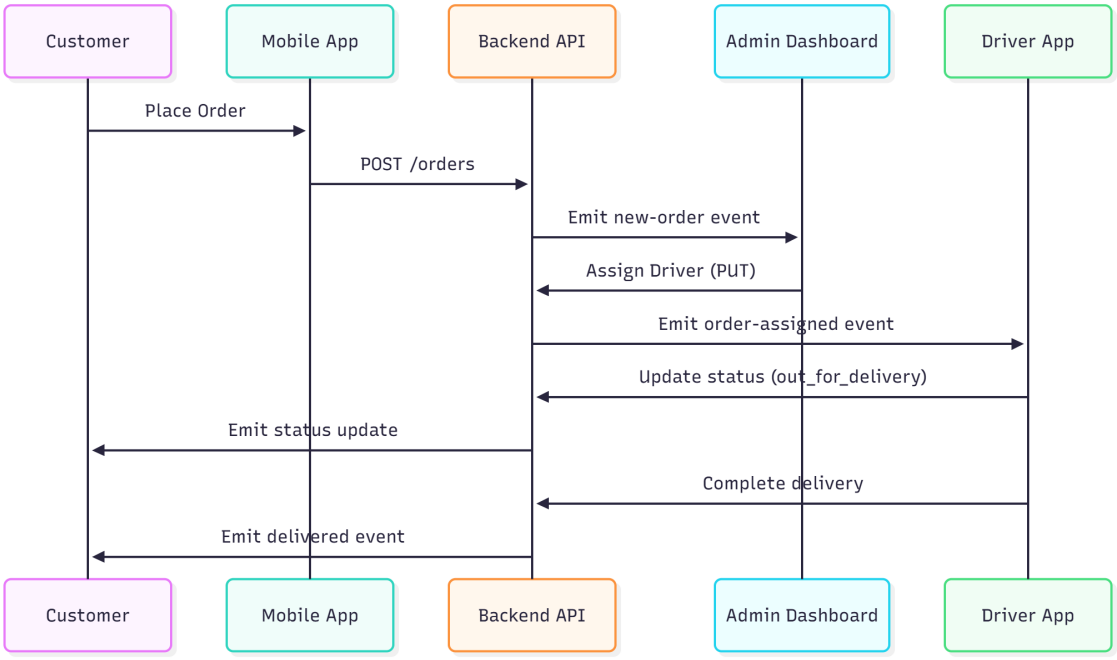

Figure 4.4: Sequence Diagram for Order Assignment

4.4.3 Activity Diagram – Customer Refill Flow

This activity flow captures the steps a customer follows to initiate and complete a refill request.

- View tank status.

- Decide to order if level is low.
- Place order.
- Order assignment.
- Delivery and confirmation.

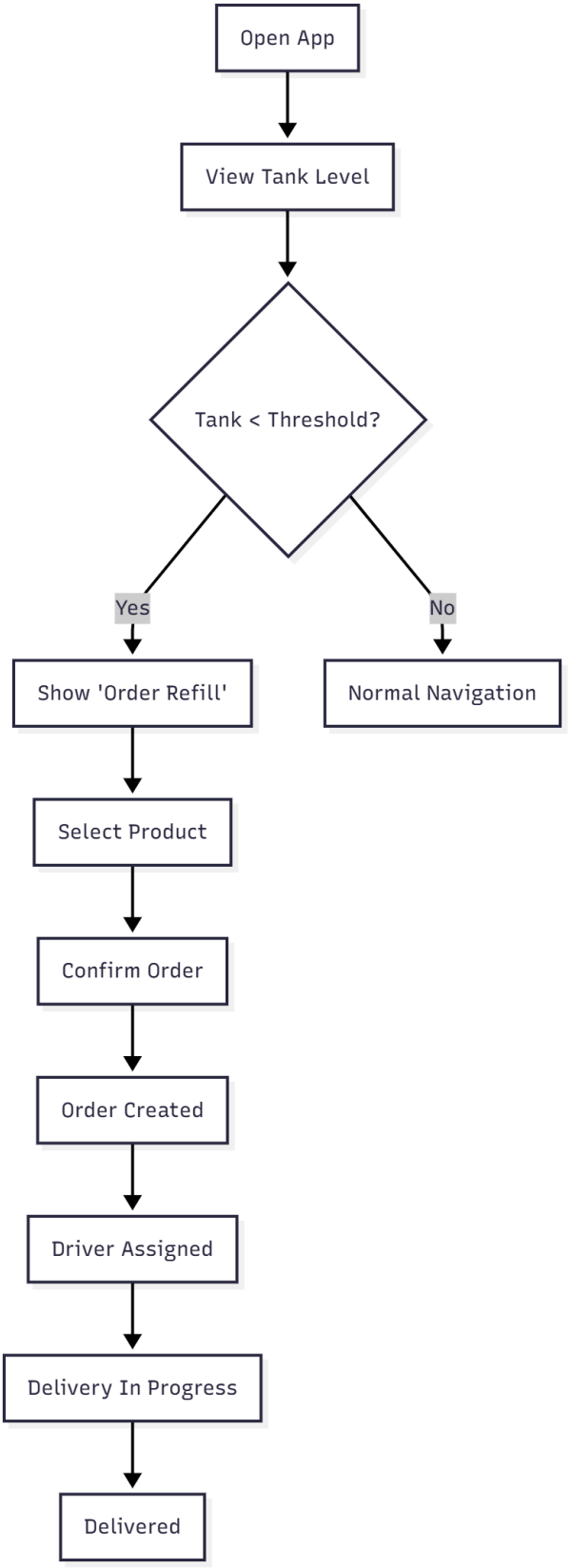

Figure 4.5: Activity Diagram – Customer Refill Flow

Chapter 5

System Implementation

The implementation of the **Urban Waterz** system spans multiple components working together to deliver a complete IoT-enabled water tank monitoring and tanker dispatch platform. This zero section provides an introductory overview of the technologies, layers, and architectural structure implemented throughout Chapter 5.

The Urban Waterz system follows a modular architecture consisting of four major layers:

- **Firmware Layer (ESP32)** – Responsible for acquiring water tank data through ultrasonic and DHT sensors, formatting it into JSON, and transmitting it securely to AWS IoT Core over MQTT.
- **Backend Layer (Node.js/Express + MongoDB)** – Handles authentication, order lifecycle management, driver queueing, IoT data ingestion, and exposes REST APIs as well as real-time communication via Socket.IO.
- **Mobile Application Layer (Expo React Native)** – Provides separate flows for customers and drivers, enabling tank-level monitoring, order placement, live driver tracking, and driver delivery operations.
- **Web Application Layer (React + Tailwind)** – An admin-oriented dashboard offering system-wide visibility over users, drivers, orders, IoT data, and operational analytics.

5.1 Technology Stack

The system uses modern development tools selected for scalability, reliability, and performance.

- **Firmware:**
  - PlatformIO (ESP32), C++
  - Libraries: WiFiManager, PubSubClient, ArduinoJson, DHT sensor library
- **Backend:**
  - Node.js, Express
  - MongoDB, Mongoose
  - Socket.IO
  - AWS IoT SDK, bcrypt, JSON Web Token (JWT)
- **Mobile App:**
  - React Native [5], Expo
  - Expo Router for navigation
  - Axios for HTTP requests
  - Socket.IO client
  - react-native-maps for location visualization
- **Web App:**
  - React, React Router
  - Tailwind CSS
  - Axios
  - (Optional) Socket.IO client

5.2 Firmware Implementation

The firmware enables data collection and secure communication between the IoT device and the cloud.

5.2.1 Setup and Configuration

- `platformio.ini` specifies:
  - Board: `esp32doit-devkit-v1`
  - Framework: arduino
  - Monitor baud rate
  - Library dependencies
- `config.h` defines:
  - Sensor pins (DHT, ultrasonic, soil moisture)
  - MQTT topics (`AWS_IOT_PUBLISH_TOPIC`, `AWS_IOT_SUBSCRIBE_TOPIC`)
  - Time zone
- `secrets.h` contains:
  - WiFi SSID & password
  - AWS IoT endpoint
  - Device certificate and private key

5.2.2 Main Loop Logic

The firmware follows a non-blocking loop structure to maintain stable network and sensor operations. High-level algorithm:

1. **Initialization:**

- Start serial communication
- Connect to Wi-Fi using WiFiManager
- Connect to AWS IoT using `connectAWS`
- Initialize sensors using `initSensor`
- Synchronize time using `NTPConnect`

2. **Loop:**
   - Maintain MQTT connection (`clientLoop()`)
   - Attempt reconnection if disconnected
   - Every 5 seconds:
     - Read sensor values (distance, optionally temperature/humidity)
     - Build JSON message with humidity, temperature, distance, timestamp
     - Publish JSON to `AWS_IOT_PUBLISH_TOPIC`
3. **Debug:**
   - Print values to serial for logging

Note: The firmware is resilient to short connectivity issues and continuously attempts reconnection.

5.3 Backend Implementation

5.3.1 Server Entry and App Setup

The backend application initializes all required services and connections during startup.

- `server.js`:

  - Loads environment variables (`dotenv`)
  - Connects to MongoDB
  - Starts Express app on specified port
  - Attaches Socket.IO server to HTTP server
  - Initializes AWS IoT subscriber

- `app.js`:
  - Configures CORS
  - Parses JSON and URL-encoded bodies
  - Mounts routes: /api/auth, /api/orders, /api/admin/users, /api/drivers, /api/iot, /api/calibration, /api/driver
  - Defines /api/health endpoint
  - Global error handler for exceptions

5.3.2 Models

- **User schema:** Common fields plus customer and driver-specific fields, pre-save password hashing, `comparePassword` method
- **Order schema:** Items array with type, quantity, unitPrice, totalPrice, validation, pre-save hooks to compute totals and timestamps
- **IoTData schema:** humidity, temperature, distance, tankLevel, timestamp, receivedAt
- **Calibration schema:** tank_depth, tank_full_distance, createdAt
- **EarningsRecord, Notification, SupportTicket:** Support driver earnings, notifications, and support flows

5.3.3 Services

- **AuthService:** Handles registration/login, JWT issuance, credential verification
- **OrderService:** createOrder, getOrderById, getOrdersByCustomer, getAllOrders, updateOrderStatus, cancelOrder, assignDriver, emits socket events
- **DriverService:** updateDriverProfile, getAllDrivers, getDriverById, updateDriverStatus, assignOrderToDriver, completeCurrentOrder, reorderDriverQueue, removeOrderFromQueue, updateDriverLocation, getDriverStatistics
- **UserManagementService:** getAllUsers, getUserById, updateUser, block/unblock, changeUserType, deleteUser, getUserStatistics, searchUsers
- **IoTDataService:** processIoTData, getLatestData, getAllData
- **SocketService:** Manages Socket.IO server, room management, and event emission

5.3.4 Controllers and Routes

- **Auth Routes:** POST /register, POST /login, GET /profile, PUT /profile, PUT /change-password
- **Orders Routes:** GET /products, POST /, GET /my-orders, GET /:orderId, PUT /:orderId/status, PUT /:orderId/cancel, PUT /:orderId/assign-driver, GET /admin/statistics
- **Drivers Routes:** Admin-only and driver-specific endpoints
- **Driver-App Routes:** /api/driver for driver login, profile, notifications, support tickets, settings, earnings
- **IoT Routes:** GET /latest, GET /all, GET /status, POST /connect

- **Calibration Routes:** GET /, POST /
- **Admin User Management Routes:** Full CRUD and block/unblock with statistics and search

5.4 Mobile Application Implementation

5.4.1 Navigation and Structure

The application is structured to support modular navigation and role-based access.

- Uses Expo Router with segments:
  - main – customer area
  - driver – driver area
  - auth – login/register/OTP screens
- Shared components: Header with drawer toggle and notification icon, global styles in `commans/style.tsx`

5.4.2 Key Screens

- **Auth Screens:** Login, Signup, Forgot Password, OTP verification
- **Customer Screens:** Tank monitoring using `getLatestIoTData` and `getAllIoTData`, orders with active/history tabs, detailed order view
- **Driver Screens:** Orders list, order detail/status update, earnings summary, support ticket creation

5.4.3 Use of Hooks and Services

- `useSocket` hook: wraps socket service, exposes connection status, event subscriptions, re-joins rooms automatically

- `orderAPI`, `iotAPI`: encapsulate HTTP interactions with typed responses (TypeScript)

5.5 Web Application Implementation

5.5.1 Project Structure

The web application is organized by functionality to ensure maintainability and scalability

- `src/App.jsx`: main app component
- `src/router.jsx`: defines routes (/login, /register, /dashboard, /admin, etc.)
- `src/context/AuthContext.jsx`: manages auth state
- `src/services/api.js`: wraps Axios with base URL and interceptors
- `src/pages`: Home, Login, Register, Dashboard, Admin Dashboard

5.5.2 Admin Dashboard Features

- **Authentication:** login form posting to backend, stores token in local storage, attaches to Axios headers
- **Main Dashboard:** summary cards (total orders, revenue, active users), charts (orders by status, per day)
- **User Management Page:** table of users, filters/search, action buttons (block, unblock, change type, delete)
- **Orders Management Page:** table of orders, status, driver info, admin actions

- **Drivers Management Page:** list of drivers, status, queue length, driver detail view

5.6 Security and Error Handling

- **JWT Authentication:** includes id, email, userType; validated in HTTP and socket middleware
- **Password Security:** hashed with bcrypt, never returned in responses
- **RBAC Rules:** protected routes require correct user type/permissions; admin operations restricted
- **Error Handling:** global error handler logs and returns standardized responses; database and IoT connection failures handled gracefully

Chapter 6

System Testing and Evaluation

This chapter presents the testing, evaluation, and validation activities performed for the **Urban Waterz** system. The goal of this zero section is to provide an introductory overview of the testing strategy, methodologies, and evaluation approaches used to ensure that all components of the system function reliably and meet the intended requirements.

The Urban Waterz platform consists of interconnected modules, including the ESP32 IoT firmware, the Node.js backend with real-time services, the React Native mobile application, and the React-based web administration dashboard. Due to the multi-layered and real-time nature of the system, testing was conducted at several levels to verify correctness, robustness, usability, and performance.

Testing and evaluation activities covered the following areas:

- **Functional Testing** — Validating core features such as user authentication, order creation, driver assignment, IoT data processing, role-based access control, and real-time communication via Socket.IO.
- **Integration Testing** — Ensuring seamless interaction among components, including firmware-to-cloud communication, backend-to-mobile data flows, and admin dashboard operations.
- **System Testing** — Executing complete user scenarios such as tank monitoring, placing orders, driver delivery flows, and admin oversight.
- **Performance Evaluation** — Measuring delays between IoT sensor updates and UI display, real-time event propagation, and API response behavior under normal usage.

- **User Evaluation** — Gathering feedback from sample customers, drivers, and admin users to assess usability and overall user experience.
- **Limitations Assessment** — Identifying constraints in the current implementation, including hardware limitations, single-tank architecture, absence of integrated payments, and routing optimization gaps.

The following sections in this chapter provide detailed descriptions of each testing category, specific test cases, observed system behavior, user feedback outcomes, and known limitations encountered during evaluation.

6.1 Testing Strategy

A combination of black-box and white-box testing approaches was employed:

- **Unit Testing:**
  - Candidate functions: tankLevel calculation, order status validation, driver queue operations
- **Integration Testing:**
  - Use Postman or Insomnia to test API endpoints end-to-end with MongoDB and IoT subscriber
- **System Testing:**
  - Run mobile and web clients together with backend and firmware
  - Test complete scenarios: customer sees tank, places order, admin assigns driver, driver delivers
- **User Testing:**
  - Ask selected users to try the app and report their experience

6.2 Test Cases

Example test cases are summarized below. Tables can be expanded for more detailed testing documentation.

6.2.1 Authentication Test Cases

- **TC-Auth-1: Register New Customer**
  - Input: Valid customer registration form
  - Expected Output: HTTP 200/201, JSON with `success: true`, `userType: "customer"`, valid JWT
- **TC-Auth-2: Login with Wrong Password**
  - Input: Correct email, wrong password
  - Expected Output: HTTP 401, message: "Invalid credentials"

6.2.2 Order Flow Test Cases

- **TC-Order-1: Place Order**
  - Prerequisite: Logged-in customer
  - Input: Items array with large_tanker and water_bottles, valid address
  - Expected Output: Order created with status = pending, totals computed correctly, `new-order` socket event observed in admin client
- **TC-Order-2: Invalid Status Transition**
  - Input: Attempt to change delivered order back to pending
  - Expected Output: HTTP 400 (Bad Request), error message describing invalid transition

6.2.3 IoT Data Test Cases

- **TC-IoT-1: Process Valid Sensor Reading**
  - Input: JSON message from AWS IoT with distance, humidity, temperature
  - Expected Output: IoTData record created with correct tankLevel, `/iot/latest` returns new record
- **TC-IoT-2: No Calibration Set**
  - Precondition: Calibration collection is empty
  - Input: Incoming sensor reading
  - Expected Output: Error logged: "Calibration values not set", IoTData not stored, front-end may display friendly error

6.3 Performance Evaluation

Performance is assessed based on responsiveness and real-time behavior.

Potential metrics include:

- API response times under normal load
- Latency from sensor measurement to appearing at `/iot/latest`
- Latency from order status change until appearing on client UIs via Socket.IO events

Tests can be conducted using browser developer tools, Postman, and backend logs.

6.4 User Evaluation and Feedback

Small user groups were used to collect feedback:

- **Customers:** Evaluate ease of use for tank monitoring and order placement
- **Drivers:** Evaluate clarity of orders and usefulness of earnings view
- **Admins:** Evaluate dashboard and management tools

Qualitative comments and suggestions were recorded.

6.5 Limitations

Certain constraints affect scalability and scope at the current stage.

- Single IoT node and single tank scenario; multi-tank support not yet implemented
- No integrated payment solution (assumes cash-only)
- Limited formal automated tests
- Routing optimization for multiple drivers not implemented

Chapter 7

Conclusions and Future Work

This chapter provides a summary of the overall outcomes of the **Urban Waterz** project and highlights its major contributions along with potential avenues for future enhancement. The Urban Waterz system integrates IoT-based water tank monitoring with a complete tanker dispatch and delivery workflow, combining firmware, backend services, mobile applications, and an administrative web dashboard into a unified platform.

The core purpose of this zero section is to introduce the concluding analysis presented in this chapter and to contextualize the project's final results. The implementation successfully demonstrates how real-time IoT data, mobile service workflows, and administrative oversight can be merged to improve water delivery operations, enhance user experience, and reduce uncertainty in water availability.

The conclusion chapter covers three main components:

- **Conclusions** — A summary of the system's achievements, functional completeness, and its ability to meet the objectives defined in the early phases of the project. It reflects on the successful integration of sensing, communication, dispatch management, and user interfaces.
- **Contributions** — A detailed outline of the technical and practical innovations produced by the project, including the calibrated tank-level computation model, real-time driver assignment mechanism, multi-role system architecture, and full-stack implementation from firmware to admin dashboards.

- **Future Work** — Identification of features and research directions that can extend Urban Waterz into a more scalable and commercial system. These include integration of digital payment methods, predictive analytics based on historical IoT data, route optimization for drivers, multi-site and multi-tank support, and further automation in testing and system validation.

The subsequent sections of this chapter elaborate on these aspects, providing a comprehensive reflection on the project's outcomes and a roadmap for continued development and enhancement.

7.1 Conclusions

Urban Waterz demonstrates a full end-to-end IoT-enabled water management system:

- Successfully measures tank level and presents it to users
- Offers a complete ordering, dispatch, and tracking workflow
- Effectively uses real-time communication to keep all parties informed
- Structures data and business logic to support future growth

This project bridges the gap between simple tank monitoring tools and generic delivery platforms by integrating both domains into a single system.

7.2 Contributions

Key contributions of this final year project include:

- A complete architecture for IoT-based water tank monitoring and tanker dispatch

- A detailed role-based access control model for customers, drivers, and admins
- A driver queue and assignment mechanism integrated with real-time mobile updates
- Calibration-based tank-level computation that can be tuned for different tanks
- A prototype implementation spanning firmware, backend, mobile, and web layers

7.3 Future Work

Potential directions for future enhancements:

- **Payment Integration:**
  - Integrate credit/debit cards or local payment gateways
  - Link with `EarningsRecord` for full financial tracking
- **Predictive Analytics:**
  - Use historical IoT data and orders to predict days until empty tank
  - Suggest optimal reorder times or enable automatic scheduling
- **Route Optimization:**
  - Use mapping APIs to propose efficient delivery routes for drivers
  - Minimize travel time and fuel consumption
- **Multi-Tank and Multi-Building Support:**
  - Extend data model to link IoT nodes with multiple tanks and sites
  - Provide hierarchical dashboards for each building or complex

- **Full Test Automation:**
  - Implement Jest or Mocha tests for backend
  - Use Cypress or Detox for automated UI tests

References

[1] Espressif Systems. Esp32 technical reference manual and documentation, n.d. Accessed: 2025-12-01. Cited on p. 1.

[2] Socket.IO. Socket.io documentation, n.d. Accessed: 2025-12-01. Cited on p. 3.

[3] Amazon Web Services. Aws iot core developer guide, n.d. Accessed: 2025-12-01. Cited on p. 4.

[4] HiveMQ. Mqtt client library encyclopedia: Arduino/esp32 examples, n.d. Accessed: 2025-12-01. Cited on p. 6.

[5] Meta. React native documentation, n.d. Accessed: 2025-12-01. Cited on p. 28.

[6] MongoDB. Data modeling introduction, n.d. Accessed: 2025-12-01. Cited on p. 46.

[7] Node.js Foundation. Node.js documentation, n.d. Accessed: 2025-12-01. Cited on p. 46.

Appendix A

User Manual

A.1 System Requirements

A.1.1 Backend

- Node.js 18+.
- MongoDB (local or cloud instance) [6].

A.1.2 Firmware

- ESP32 DevKit board.
- Ultrasonic sensor, DHT11 sensor.
- USB cable for flashing firmware.

A.1.3 Mobile App

- Android or iOS smartphone.
- Expo Go app (for development).

A.1.4 Web App

- Modern browser (Chrome, Firefox, Edge).

A.2 Installation Steps

A.2.1 Backend

1. Install Node.js [7] and MongoDB.
2. Navigate to backend directory: `cd backend`.

3. Install dependencies: `npm install`.
4. Copy `.env.example` to `.env` and configure environment variables:
    - $\texttt{MONGODB}_{U}RI$,
5. Start the development server: `npm run dev`.

A.2.2 Firmware

1. Open the firmware folder in VS Code with PlatformIO.
2. Edit `secrets.h` to include:
    - WiFi SSID and password.
    - AWS IoT endpoint and certificates.
3. Connect ESP32 via USB.
4. Build and upload the project using PlatformIO.

A.2.3 Mobile App

1. Navigate to mobile directory: `cd mobile`.
2. Install dependencies: `npm install`.
3. Set `config.ts` with:
    - `apiUrl` pointing to backend
    - `backendUrl` for Socket.IO (
4. Start Expo dev server: `npm run dev`.
5. Scan QR code using Expo Go on your phone or run on emulator.

A.2.4 Web App

1. Navigate to web app directory: `cd webapp/web-app`.
2. Install dependencies: `npm install`.
3. Start development server: `npm run dev`.
4. Open the URL shown in the console (e.g., [`http://localhost:5173`](http://localho

A.3 Basic Usage

A.3.1 Customer

1. **Register Login**
    - Open the mobile app.
    - Choose ‘Sign Up” to create a new customer account.
    - Login using email and password.
2. **View Tank Level**
    - Navigate to the Tank Monitoring tab.
    - Press Refresh Data if needed.
    - Observe current tank level (%) and approximate liters.
3. **Place an Order**
    - From tank screen or orders section, choose ‘Order Refill” or ‘New Order”.
    - Select product type (large/small tanker, bottles) and quantity.
    - Confirm address and submit.
4. **Track Orders**
    - Go to Orders tab.
    - Active orders show live statuses and, if assigned, driver info and location.
5. **Cancel Order**
    - Open order details while status is still pending or confirmed.
    - Tap ‘Cancel Order” and confirm.

A.3.2 Driver

1. Login to driver area.
2. View Current Order and Queue.

3. Tap order to see details and map location.
4. Press ‘Out for Delivery” when starting, and ‘Complete Delivery” when done.
5. Check Earnings and Support sections as needed.

A.3.3 Admin

1. Login at admin web portal.
2. Open Dashboard to see key metrics (orders, users, IoT status).
3. Use Users section to manage customer and driver accounts.
4. Use Orders section to view, filter, and assign orders.
5. Use Drivers section to monitor driver statuses and queues.
6. Use IoT section to inspect tank data trends.

Appendix A

Additional Diagrams

This appendix contains supplementary diagrams to support the system design and workflows.

A.1 UML Class Diagram

A.2 Sequence Diagrams

A.2.1 IoT Data Processing

A.2.2 Support Ticket Flows

A.3 State Transition Diagrams

A.3.1 Order Status

A.3.2 Driver Status

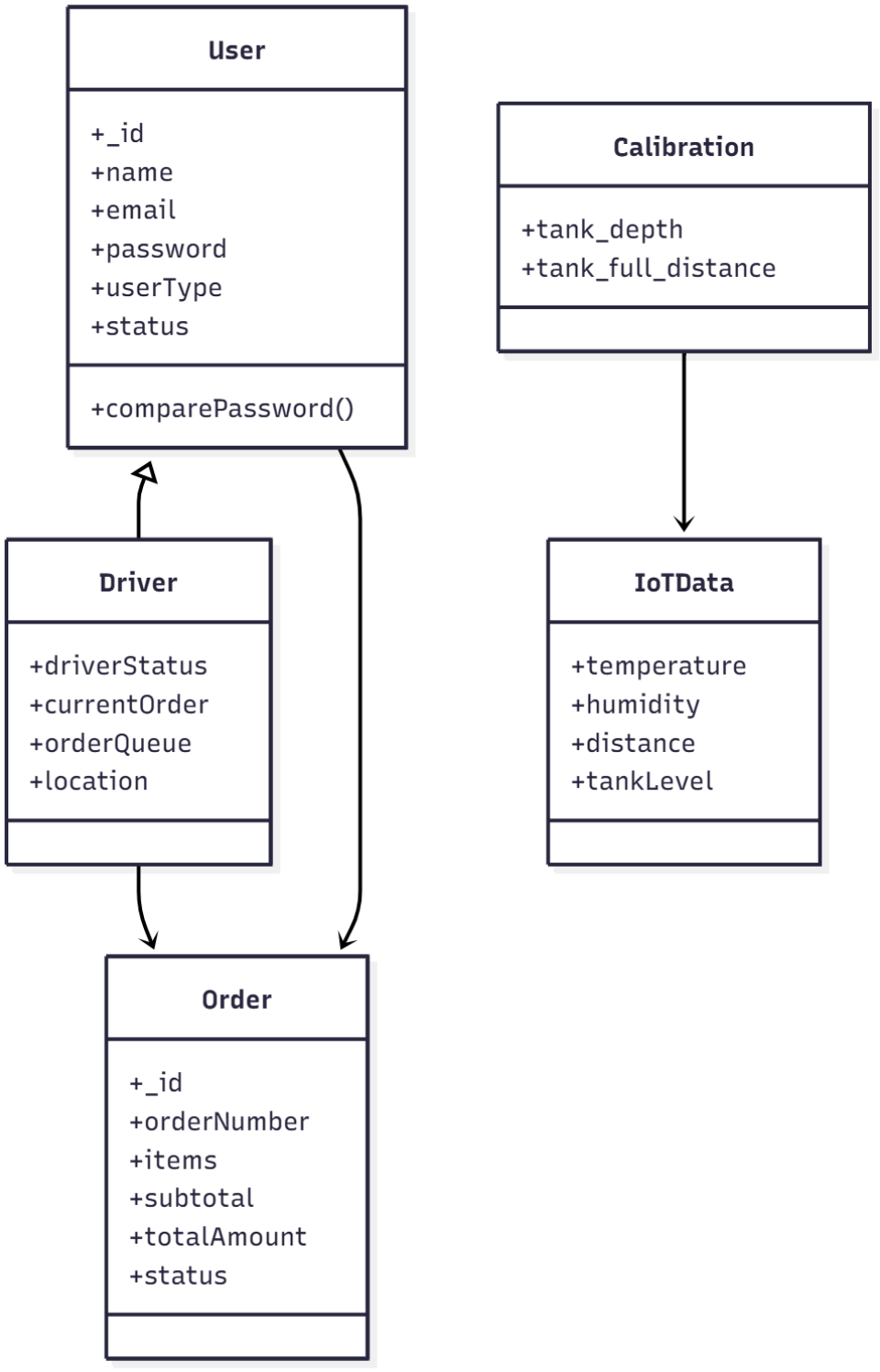

Figure A.1: Full UML Class Diagram for all models.

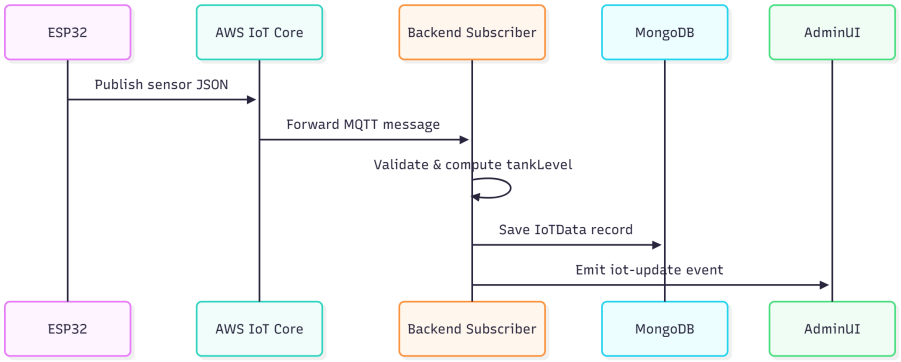

Figure A.2: Sequence diagram showing IoT data flow from sensors to backend and apps.

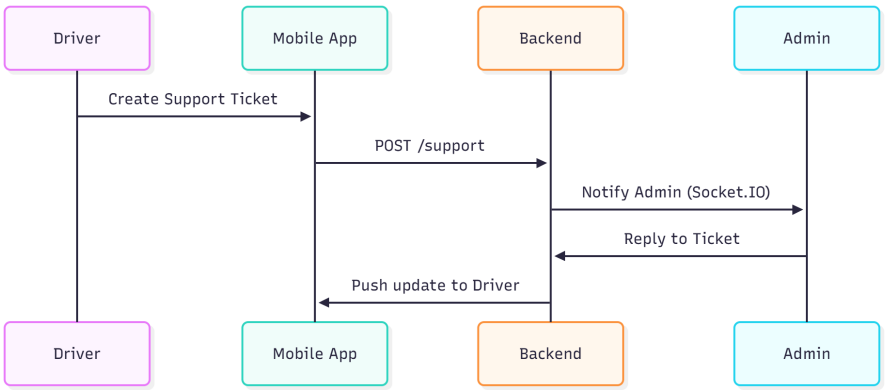

Figure A.3: Sequence diagram for customer support ticket creation, assignment, and resolution.

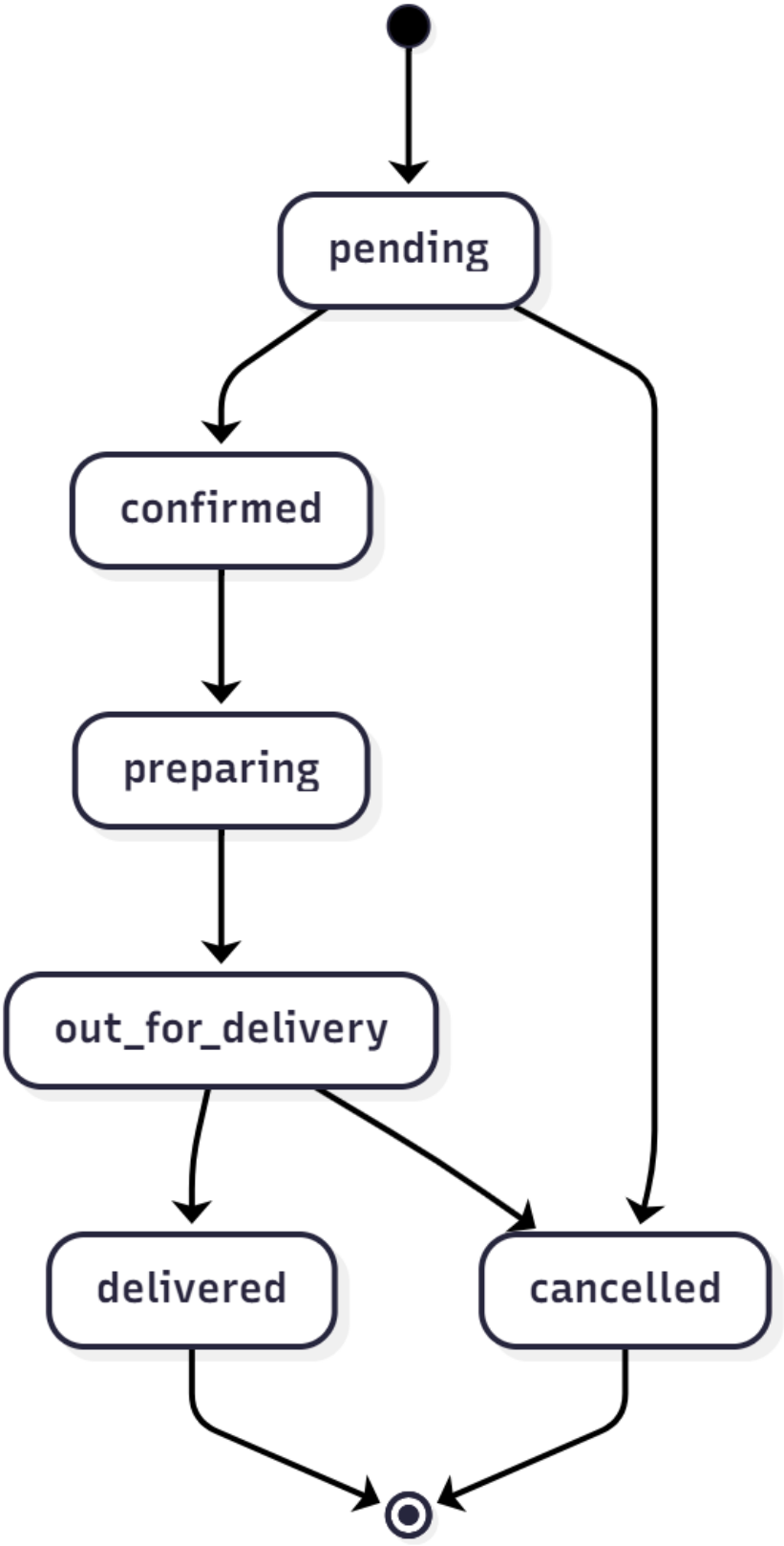

Figure A.4: State transition diagram illustrating the order lifecycle: Pending → Confirmed → Out for Delivery → Completed → Cancelled.

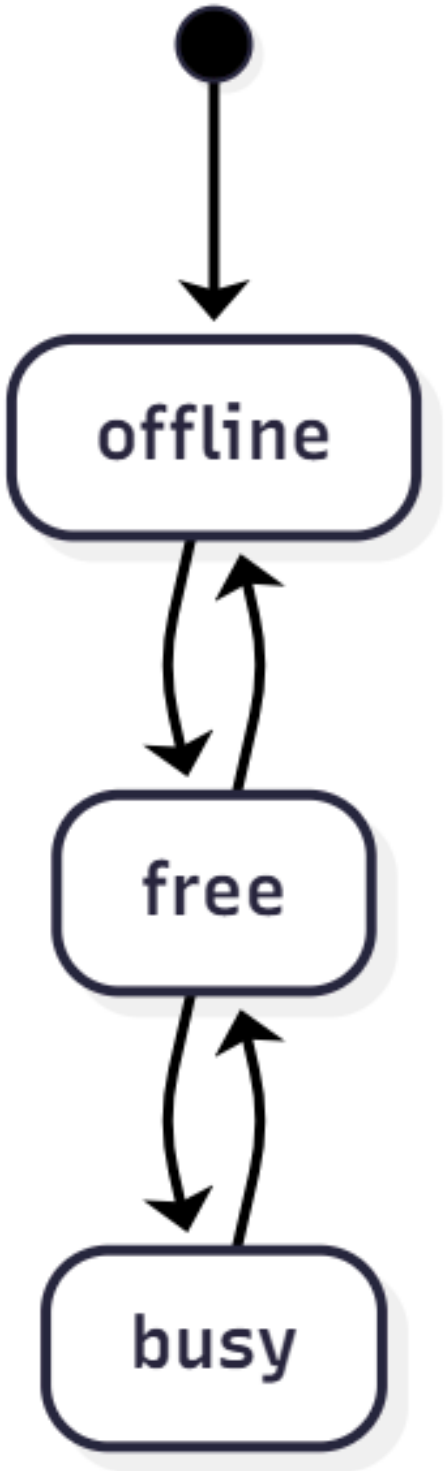

Figure A.5: State transition diagram showing driver availability states: Offline → Available → On Delivery → Break → Offline.

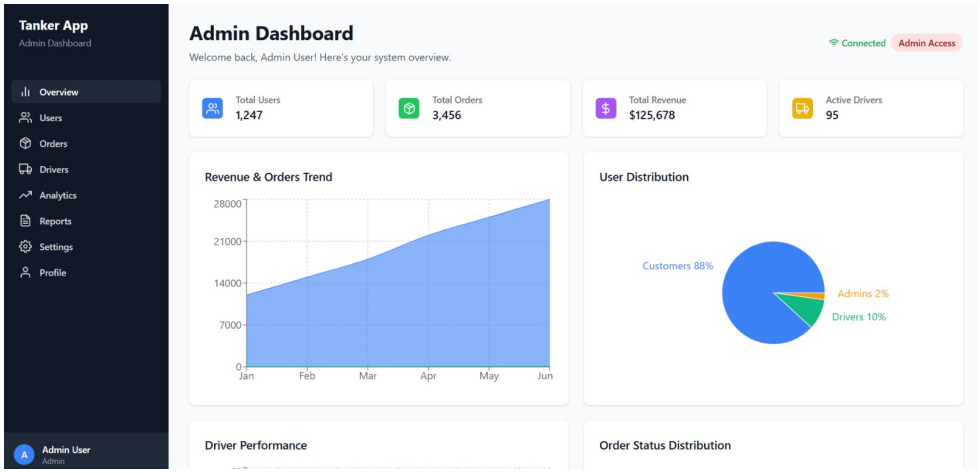

Figure A.6: Admin Overview

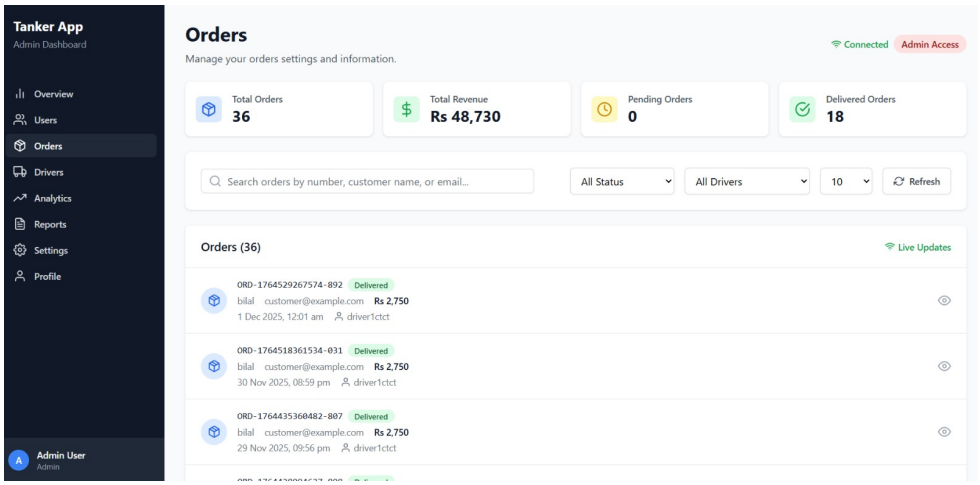

Figure A.7: Admin View Orders

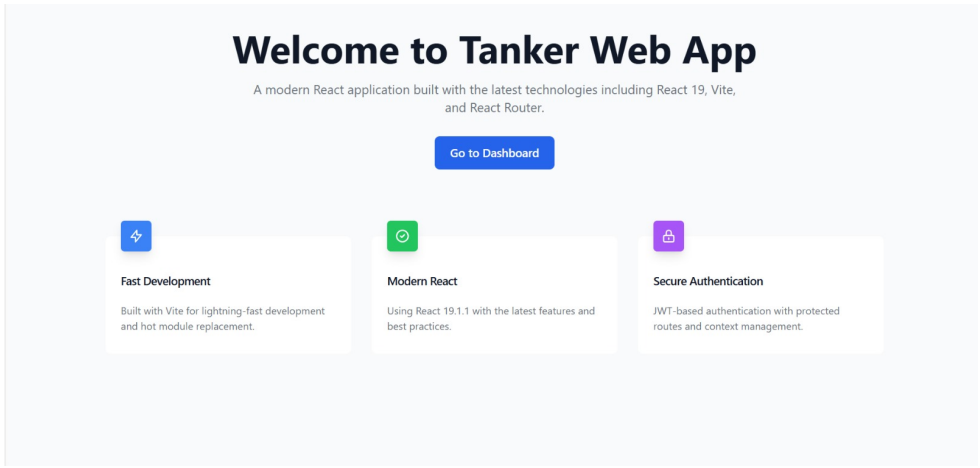

Figure A.8: Admin HomePage

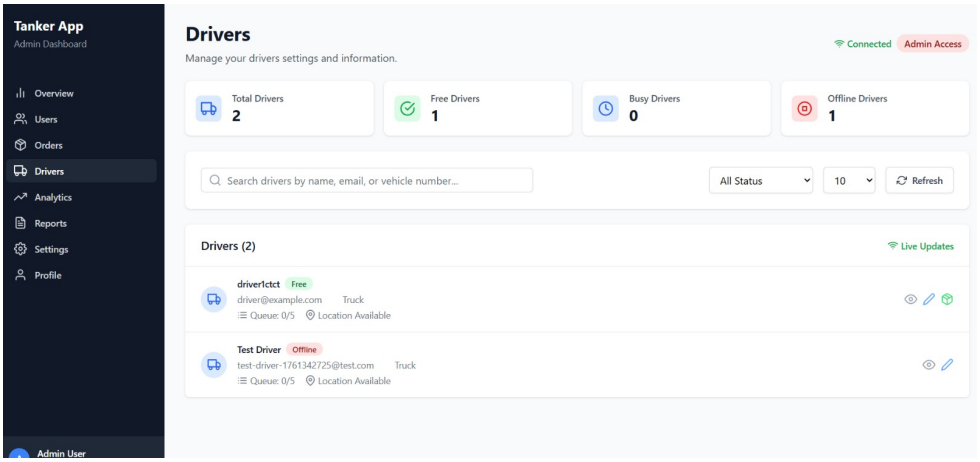

Figure A.9: Admin Driver Page

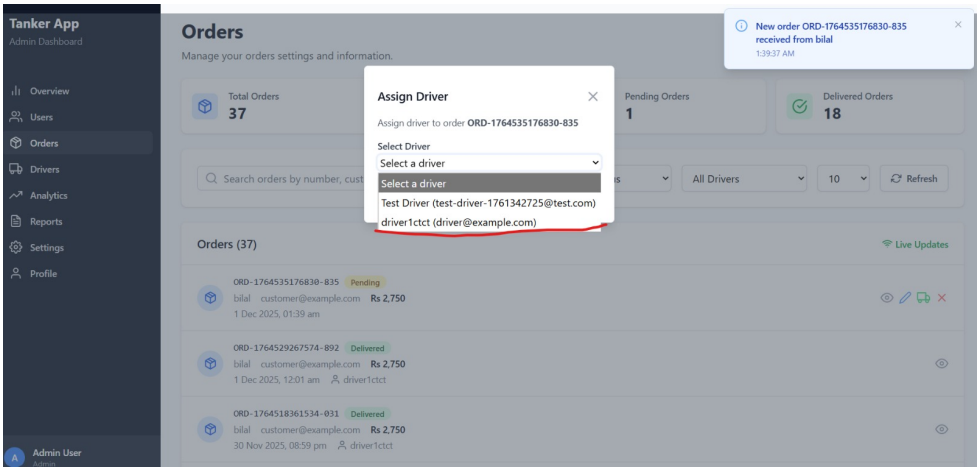

Figure A.10: Admin Assigning Driver

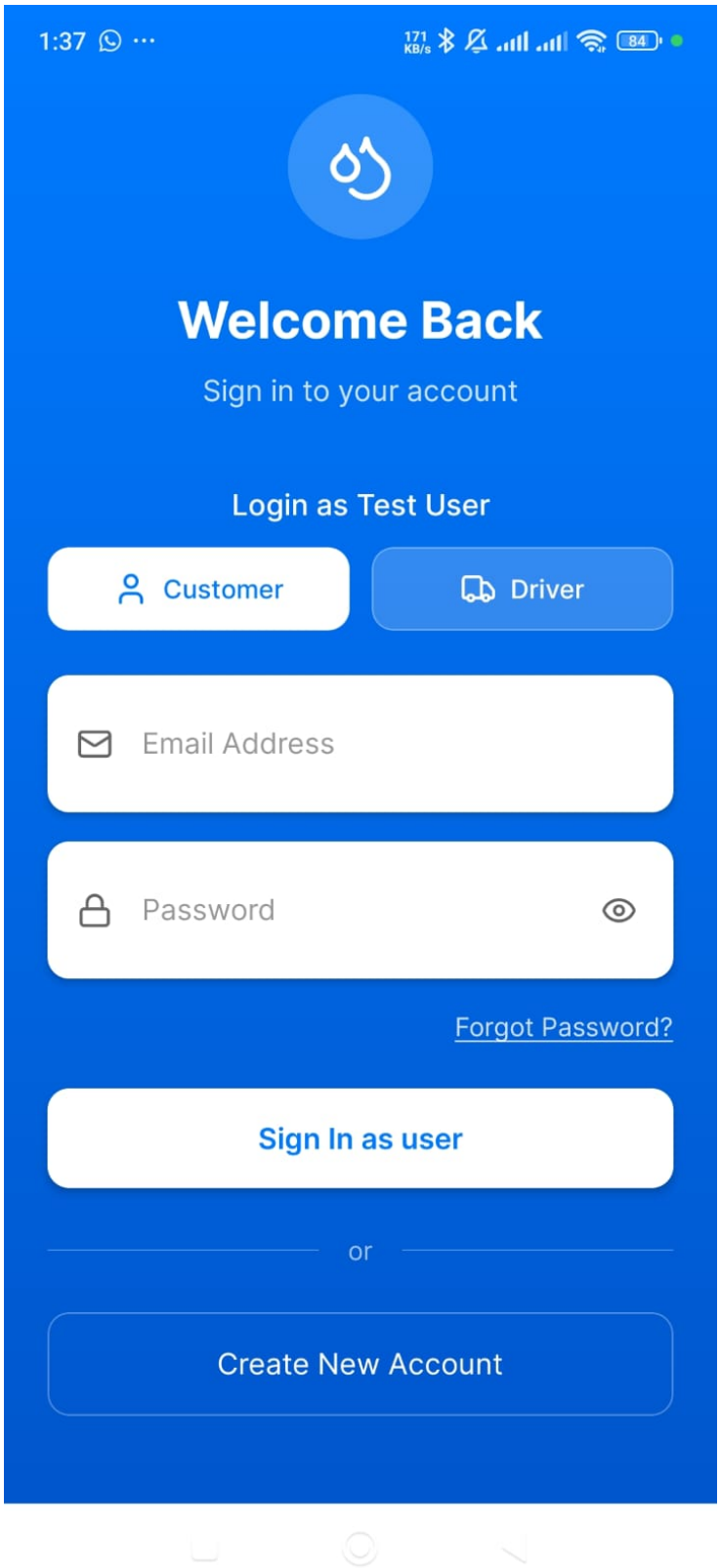

Figure A.11: User Login

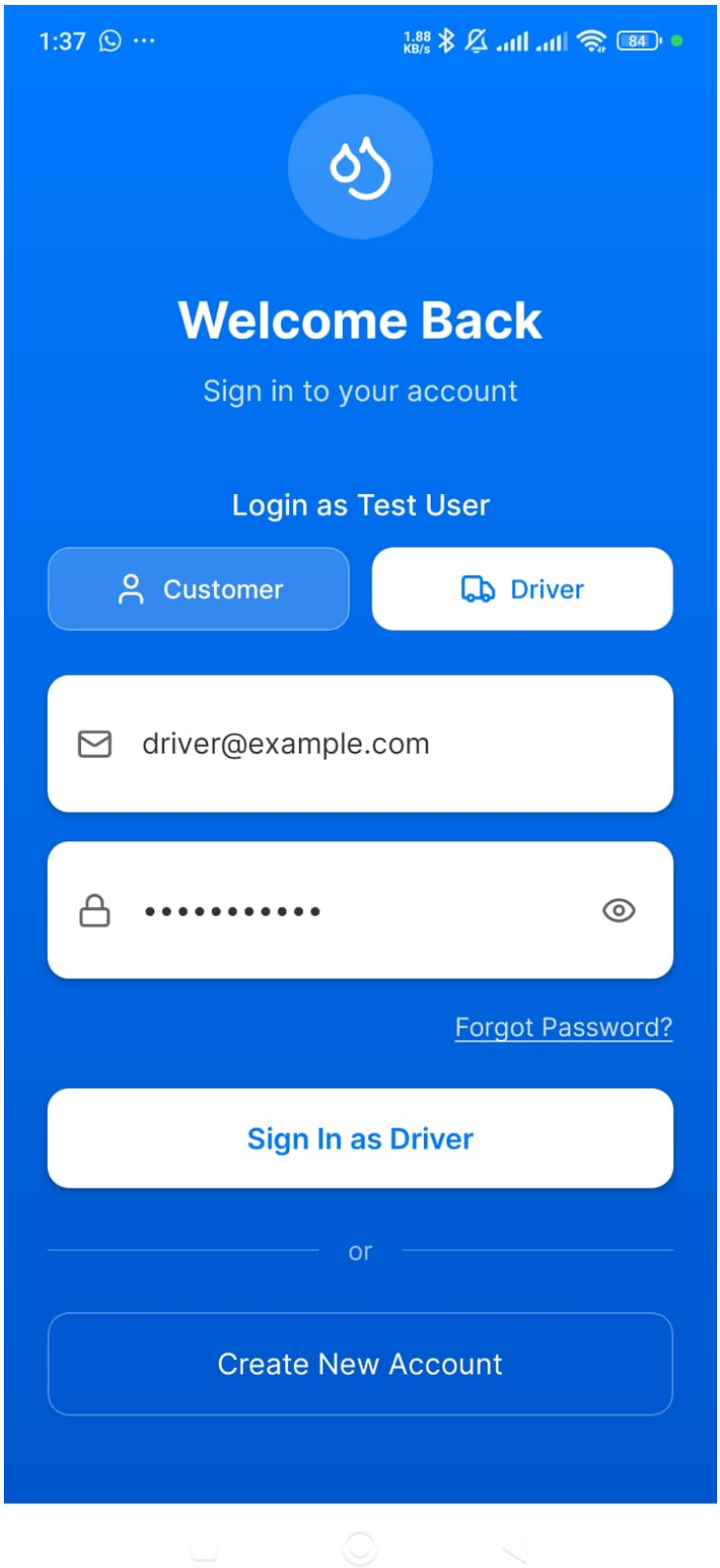

Figure A.12: Driver Login

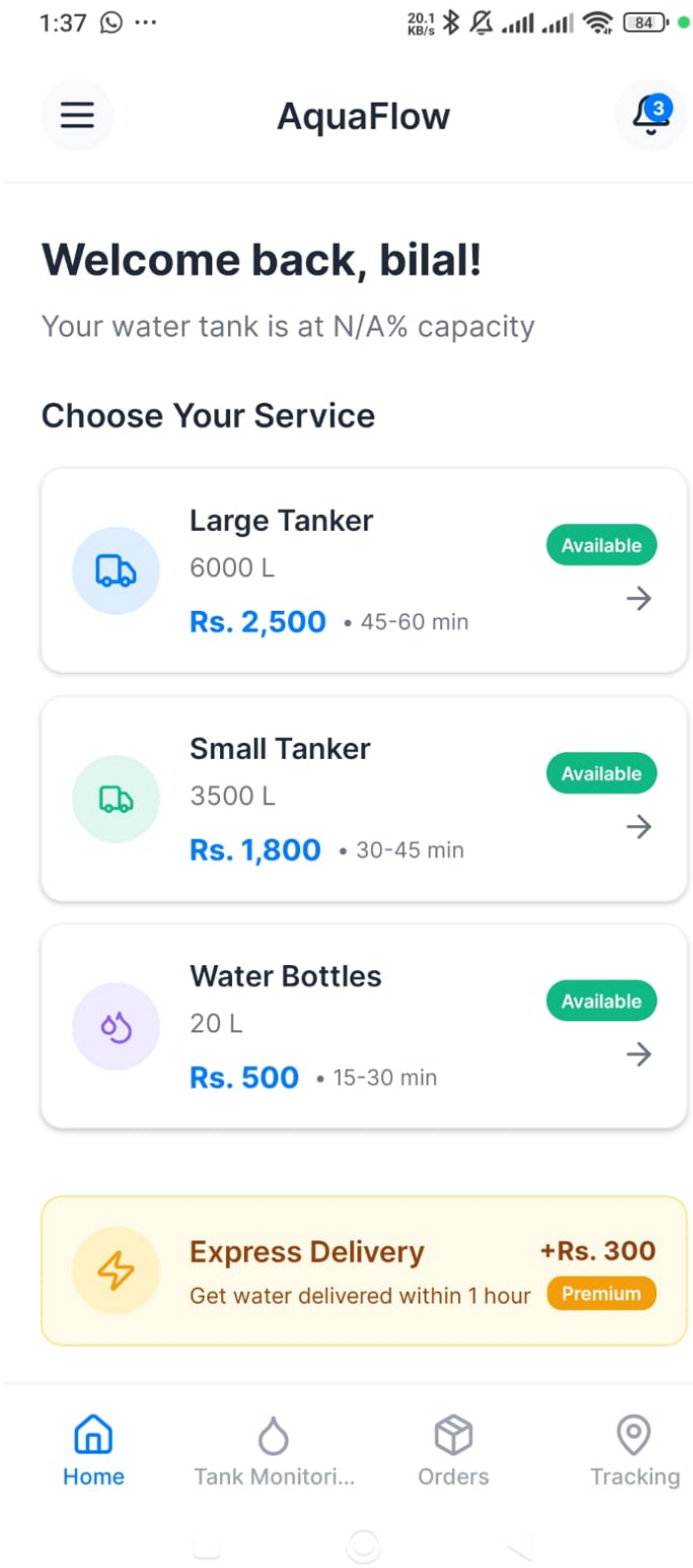

Figure A.13: User HomePage